A Type System for Data-Centric Synchronization

Mandana Vaziri[‡] Frank Tip[‡] Julian Dolby[‡] Christian Hammer[†] Jan Vitek[†,‡]

[‡] IBM T.J. Watson Research Center [†] Computer Science Dept., Purdue University
mvaziri|ftip|dolby@us.ibm.com cjhammer|jv@cs.purdue.edu

Abstract. Data-centric synchronization groups fields of objects into *atomic sets* to indicate they must be updated atomically. Each atomic set has associated *units of work*, code fragments that preserve the consistency of that atomic set. We present a type system for data-centric synchronization that enables separate compilation and supports atomic sets that span multiple objects, thus allowing recursive data structures to be updated atomically. The type system supports full encapsulation for more efficient code generation. We evaluate our proposal using AJ, which extends the Java programming language with data-centric synchronization. We report on the implementation of a compiler and on refactoring classes from standard libraries and a multi-threaded benchmark to use atomic sets. Our results suggest that data-centric synchronization enjoys low annotation overhead while preventing high-level data races.

1 Introduction

Writing correctly synchronized concurrent programs is challenging, and inconsistent results may be computed when programmers make mistakes. Traditional approaches to concurrent programming have an operational, control-centric, flavor. Programmers must understand where to put invocations to the particular concurrency-control primitives provided by their programming language or operating system of choice. Forgetting to protect some path may result in intermittent software faults that are frustratingly difficult to identify and eradicate. *Data-centric synchronization* [21] is a declarative approach to concurrency control. Instead of focusing on the flow of control, programmers identify sets of memory locations that share some consistency property and group those locations in *atomic sets* that will be updated atomically. The programmer need not specify where or what kind of synchronization operations to insert; instead, each atomic set has an associated set of *units of work*, code fragments that must preserve the consistency of that atomic set. Synchronization code is automatically generated by a compiler which is free to choose where and what type of synchronization to insert. Such a declarative approach allows changing the concurrency-control mechanism, e.g., going from standard locks to read/write locks or even to transactional memory, without changing the program's source code.

In our previous work, we relied on static whole-program program analysis to infer where synchronization operations should be placed in order to ensure that units of work are serializable from the perspective of each atomic set, a property we call *atomic-set serializability*. We explored the integration of atomic sets in the context of an object-oriented language; in this language, classes declare atomic sets, the elements

of an atomic set are a subset of the fields of its declaring class and its subclasses, and the units of work are the methods of the class. Experiments demonstrated that atomic sets require fewer annotations than implementations based on synchronized blocks in Java while eliminating known concurrency-related errors [24,12]. However, while the approach appeared promising, its reliance on whole-program analysis was a limitation that hindered adoption. Whole-program analysis is prohibitively expensive for large code bases. Moreover, dynamic loading, native methods and reflection are integral parts of the Java platform. Disallowing them is completely unrealistic. Our previous work was also limited by its lack of support for atomic sets spanning multiple objects, which led to inefficient code.

This paper overcomes the limitations of prior work. Specifically, we present a variant of the atomic sets model of [21] which introduces a new mechanism for constructing atomic sets that span multiple objects and for internal objects that provide strong encapsulation for data whose concurrency is managed externally. The new approach obviates the need for whole-program analysis with a type system that guarantees that any well-typed program is atomic-set serializable, which intuitively means that all operations performed on locations that belong to an atomic set are serializable. To empirically evaluate the applicability of our ideas on real-world code, we define a backwards-compatible extension of Java called AJ and implement it within the Eclipse development environment. We then refactor classes from the Java Collections Framework and a version of the SPECjbb performance benchmark into AJ, and measured annotation overhead and performance. AJ requires roughly 6.2 annotations per thousand lines of source code (KLOC) for SPECjbb, and 40 annotations per KLOC for collection classes. These annotations replace traditional synchronization constructs on blocks and methods. In terms of performance, the AJ versions of SPECjbb achieve a throughput of between 54.3% and 80.4% of that of the original Java implementation. The slowdown is explained in part by the fact that the original code is undersynchronized[1], and that our prototype compiler is a naive translator, and much work on optimization remains to be done. Therefore, we consider these results an indication that our approach is capable of generating code with acceptable performance while providing a correctness guarantee that Java's current synchronization mechanism does not offer. In summary, we make the following contributions:

- A data-centric approach to synchronization that permits separate compilation, multi-object atomic sets and strongly encapsulated objects.
- A formalization of the type system for a core calculus and a proof that any well-typed program is atomic-set serializable.
- A prototype implementation in a mainstream object-oriented language and an integration with a development environment.
- An empirical evaluation on 18,200 lines of Java code including widely used libraries and a concurrent application.

Code and additional information can be found at http://sss.cs.purdue.edu/projects/aj.

[1] It is hard to say with certainty that SPECjbb is buggy, since it is a benchmark and its own self-checking succeeds even with no synchronization at all. However, we observed clearly related operations, some of which were synchronized and others which were not. It appeared that this synchronization was inconsistent.

2 Background and Influences

This paper builds on the atomic set programming model of Vaziri, Tip and Dolby [21]. That work introduced a notion of problematic interleaving scenarios and then used it to define a correctness criterion, atomic-set serializability, which rules out high-level data races in code manipulating atomic sets. Subsequent work explored how to detect concurrency-related errors based on this criterion (statically in [17] and dynamically in [12]). Atomic sets share characteristics with data groups [18] and regions [10] which group mutable fields to enable modular verification and reasoning about program transformations. Like atomic sets, regions and groups may be extended in subclasses, but unlike atomic sets, both are hierarchical and regions overlap. Another data-centric approach was proposed in [4], with a sketch of a possible transactional memory implementation. Atomic sets can also be viewed as a generalization of Hoare monitors [15] to multiple objects. While atomic sets currently lack a counterpart for condition variables, they add the unitfor construct to temporarily merge distinct atomic sets.

Data-centric concurrency control is but one alternative to explicit locking. Transactional memory [14] approaches concurrency control from a database angle: Certain code fragments are specified to execute atomically, and it is up to the implementation to enforce mutual exclusion. While programmers need not worry about which data will be accessed in a transaction, they still have to identify where to place atomic sections. Another way to avoid explicit locking is to perform lock inference. Like transactional memory, programmers must annotate programs with atomic sections, but instead of relying on a transactional memory mechanism, static analysis is used to determine which locks to acquire [5,19]. While more efficient than transactions, as there is no need to support abort/undo semantics, lock inference relies on whole-program information and thus can not deal with dynamic features of Java.

Type systems for atomicity and race-freedom are another influence on our work. The type system of [1] guarantees the absence of data races. The general approach is to have a programmer provide redundant type annotations on top of a program with explicit lock operations. The type system thus only needs to check that the synchronization and the type annotations are consistent. In that approach, methods declare the locks they require and a guarded_by construct is used to indicate which lock protects a field. With 20 annotations per KLOC for the Java collections framework, the approach is relatively lightweight, but unlike atomic sets the programmer must add explicit synchronization to the code. Moreover, atomic-set serializability is a higher level property than data race freedom. The type system of Flanagan and Qadeer [7] guarantees atomicity, i.e., equivalence to a serial execution. As above, fields are annotated with guarded_by or write_guarded_by to indicate that (write) access to the field must be protected by a lock. Methods are annotated with atomic to indicate their atomicity and with requires to indicate which locks must be held by callers. Atomic set serializability recognizes more interleavings as correct than global serializability. Flanagan and Qadeer evaluated their type system on Java library classes and report an average of 23.3 annotations per KLOC of code. However, as in [1] and unlike atomic sets, it is assumed that the programmer has added synchronization to the code. Inference [9] reduces the annotation burden.

Our type system was influenced by ownership type systems which started out as an attempt to control the sharing of references [20] and is typically used to enforce a strong

form of encapsulation. Our treatment of internal objects is close to traditional ownership as all references to these objects are encapsulated. But unlike the early owner-as-dominator type systems [6] there is no single access point. Indeed, in order to support iterators we have loosened the restriction of a single owner and allow the elements of atomic sets that are not part of internal classes to be viewed and manipulated from the outside. The ownership type system of [3] ensures that Java-like programs are data race-free. In that work, classes are parameterized with a list of owners and methods may require that their callers hold particular locks. A simple unification-based form of local type inference is used to reduce the annotation burden. While no direct comparison is possible as the implementation of [3] is not available, we believe atomic sets have lower annotation overhead overall, and are better integrated into Java. Deadlocks can also be ruled out by ownership type systems [2] but this comes at the price of expressiveness and an increased annotation burden.

3 Data-centric Synchronization with AJ

AJ extends the syntax of the Java programming language with annotations needed to support the data-centric programming model of [21]. These annotations are summarized in Fig. 1. An AJ class can have zero or more atomicset declarations. Each atomic set has a symbolic name and intuitively corresponds to a logical lock protecting a set of memory locations. Associated with each atomic set is a set of *units of work*, code

`atomicset a`

A class or interface declaration may have multiple atomic set declarations. Atomic sets are inherited and can be referred to in subclasses.

`atomic(a)`

Annotation on instance fields and classes. A field can belong to at most one atomic set. Annotated fields can only be accessed from the this reference. When added to a class declaration, this annotation is a shorthand for placing the same annotation on all instance fields in the class and its subclasses.

`unitfor(a)`

Each method argument can be annotated by one or more unitfor annotations. When the name is omitted, the annotated method becomes a unit of work for *all* atomic sets in the parameter object.

`internal`

Annotation on class declarations which must be preserved by inheritance. The type system tracks internal objects and ensures that no reference to an internal object can leak outside of the object that constructs it.

`|a=this.b|`

Annotation on variable declarations and in constructor expressions to indicate that the atomic set a of the type of the variable or constructed object is aliased with the current object's atomic set b.

Fig. 1. Data-centric annotations in AJ.

fragments that, when executed without interruption, preserve the consistency of their associated atomic sets. AJ assumes that all non-private methods of a class are units of work for the atomic sets it declares. Given data-centric synchronization annotations, AJ infers the placement of concurrency control operations in such a way that units of work are serializable from the perspective of each atomic set, a property we call atomic-set serializability. The inferred synchronization ensures that any execution is equivalent to an execution in which, for each atomic set, its units of work occur in some serial order. One may think of a unit of work as being an atomic section [13] that is only atomic with respect to a particular set of memory locations. Accesses to locations not in the set are visible to other threads. The AJ implementation is free to choose the type of concurrency control operations and to optimize their placement. Thus, for instance, methods declared private or called through this usually do not require synchronization as their calling context has established atomicity. Methods that do not operate on locations that are within an atomic set will typically not be synchronized either. Fig. 2 shows an integer counter class with atomic increment and decrement methods.

```
class Counter {
   atomicset a;
   atomic(a) int val;
   int get() { return val; }
   void dec() { val--; }
   void inc() { val++; }
}
```

```
Counter c = new Counter();
c.inc();
c.dec();
...
```

Fig. 2. A simple counter class.

Each instance of Counter has its own instance of its atomic set a. The locations protected by the atomic sets are identified by annotating the corresponding fields with atomic (a). Atomic set declarations are inherited by subclasses, so every instance of a subclass of Counter has its own a and can add some of its fields to the atomic set. AJ requires that fields belonging to an atomic set must be accessed through the (implicit) this reference. Note that this is stronger than labeling the field private, as in Java two instances of the same class can access each others private fields.

It is often the case that an atomic set must protect fields belonging to more than one object. While it is not possible to refer directly to another object's atomic set, AJ allows merging atomic sets using *aliasing* annotations. An atomic set a in an object pointed to by a variable x may be aliased with an atomic set b in the object pointed to by this by placing the alias annotation $|\mathsf{a} = \mathsf{this.b}|$ on the declaration of x. This has the effect of merging the atomic sets in these objects. Fig. 3 shows a PairCounter class which has two integer counters and updates the difference between them. To do this it introduces a new atomic set b for the diff field, and it aliases the atomic sets of the counters with b to form a single atomic set.

There are cases where a method needs to coarsen the granularity of atomicity for some of its arguments. This is achieved by declaring additional units of work by annotating arguments with unitfor(a). If this annotation appears on some parameter p of some method m of a class D, this indicates that m is an additional unit of work for atomic set

```
class PairCounter {
   atomicset b;
   atomic(b) int diff;
   Counter|a=this.b| low = new Counter|a=this.b|();
   Counter|a=this.b| high = new Counter|a=this.b|();
   void incHigh() { high.inc();   diff = high.get()-low.get(); }
   ...
}
```

Fig. 3. Aliased atomic sets.

a of object p. Such cases, where a method is a unit of work for multiple atomic sets are treated as if the method is a unit of work for the *union* of these atomic sets. Alias annotations have a similar effect. Fig. 4 illustrates this with a transfer method which must atomically update two Counter objects with different atomic sets.

```
class Transfer {
   void transfer(unitfor(a) Counter from, unitfor(a) Counter to) {   from.dec(); to.inc();   }
}
```

Fig. 4. Adding atomic sets to a unit of work using unitfor.

For performance reasons it may be advantageous to avoid synchronization around objects that are used to implement the representation of a given data structure. This is safe only if it is guaranteed that no reference to these representation objects ever leaks to clients where it could be manipulated without synchronization. The internal annotation is used to declare a class, or interface, and all of its subclasses as being private to a data structure. Internal classes must always have their atomic sets aliased to some enclosing data structure. The AJ type system enforces encapsulation of internal classes. The example of Fig. 5 illustrates the use of internal classes. Class Cell is internal. Class Main creates an instance of Cell, aliases its atomic set and stores it in field c. The type system ensures that the Cell object will only be manipulated by the corresponding Main object.

```
internal class Cell {
   atomicset b;   atomic(b) Object val;
   Object getset(Object o) { Object old = val; val = o; return old; }
}
class Main {
   atomicset a;   final Cell|b=this.a| c = new Cell|b=this.a|();
   void set(Object o) { c.getset(o); }
}
```

Fig. 5. An internal class.

It is noteworthy to observe that the internal annotation does not change the semantics of the application; its purpose is to enable the implementation to remove some redundant synchronization operations. While it would be possible to infer this annotation, doing so would require interprocedural analysis which we avoid in this work.

3.1 Motivating Example

We present a simplified version of the LinkedList class, a representative of the Java Standard Collections framework, made thread-safe using data-centric synchronization. Fig. 6 shows the abstract class AbsList which defines the interface of all lists and a concrete list, LinkedList. The designer of the abstract list has chosen to equip it with an atomic set a which is inherited by subclasses. Within AbsList the only field that needs protection is the integer size. It is annotated atomic(a) to denote that it belongs to a. The methods of AbsList and its subclasses are the units of work for a.

```
public abstract class AbsList {
  atomicset a;
  atomic(a) int size;
  public int size(){ return size; }
  public abstract ListIterator iterator();
  public abstract void add(Object o);
  public abstract boolean
     addAll(unitfor(a) AbsList c);
  public abstract Object get(int i);
}

internal class Entry {
  atomicset b;
  atomic(b) Object elem;
  atomic(b) Entry next|b=this.b|;
  atomic(b) Entry prev|b=this.b|;
  ...
}
```

```
class LinkedList extends AbsList {
   atomic(a) Entry header|b=this.a|;
   public LinkedList() {
      header = new Entry|b=this.a|(null,null,null);
      header.next = header.prev = header;
   }
   public void add(Object o) {
      Entry newEntry|b=this.a| =
       new Entry|b=this.a|(o, header, header.prev);
      newEntry.prev.next = newEntry;
      newEntry.next.prev = newEntry;
      size++;
   }
   public ListIterator iterator() {
      return (ListIterator)
      new ListItr|c=this.a|(this,
               this.header, 0);
   }
   ... // other list methods
}
```

Fig. 6. AbsList, LinkedList and Entry classes

The addAll(unitfor(a) AbsList c) method must operate on multiple atomic sets, namely the receiver and the argument c. Logically, the list c must remain unchanged during the entire execution of addAll. By annotating parameter c with unitfor(a), we merge the atomic set a in the receiver object with the atomic set a in the argument object for the duration of the method's execution.

In LinkedList, the header field points to a doubly-linked list of Entry objects. LinkedList adds header to the atomic set of its parent class to ensure that any method accessing both header and size will have a consistent view of the fields. The above is not sufficient for the data structure to be thread-safe. It is also necessary to protect the doubly-linked list itself. This requires defining an atomic set b in class Entry to protect the fields next and prev. Furthermore, units of work for the LinkedList object must encompass the units of work for the Entry objects it refers to. This is achieved by using the alias annotation |b=this.a| to indicate that the atomic set b of the Entry object should be combined with the list's atomic set a. These annotations are placed on all allocation sites and variables

of type Entry. Similar annotations, |b=this.b|, are placed on the fields next and prev of Entry. These imply that the atomic sets b of objects pointed to by these fields are merged with the atomic set b of this. Together with the annotation on header, they cause the entire backbone of the LinkedList to be in a single atomic set. Any unit of work for the list, including its Entry objects, will be performed atomically with respect to this merged atomic set. As an optimization, Entry is declared internal. This means that the type system will guarantee that no instance of Entry can be accessed without going through the methods of LinkedList. Thus, an implementation can omit synchronization for all of Entry's methods and leave concurrency control to the list object.

Each expression in our type system potentially has alias information. If there is no alias information, this means that either the expression represents an object that has no atomic sets, or that the object is an independent object that performs its own synchronization. The type system tracks aliasing annotations and prevents, e.g., the Entry object of one linked list from ending up within another linked list. Practically this means that some types of casts are disallowed. It is allowed to cast away an alias annotation (thus losing information), but forging an alias annotation is not. For instance, the iterator() method creates an object of type ListItr (a class that is private to class LinkedList), which has an atomic set aliased to that of the linked list. This alias information is cast away in the return statement of the method.

A non-internal class such as LinkedList can be instantiated in two ways: new LinkedList() and new LinkedList|a=this.x|. The former signifies a new instance of LinkedList that is responsible for its own synchronization, while the latter means that atomic set of the new instance is the same as that of atomic set x of the current object. The latter is especially useful when defining new data structures in terms of other data structures. One could define a Stack in terms of a LinkedList; correct synchronization behavior can be achieved by having an atomic set in Stack that is aliased to the atomic set in the underlying LinkedList. This kind of compositionality is a key contribution of this paper and was not supported in [21]. For internal classes such as Entry an aliased allocation site such as new Entry|b=this.a| is the only valid instantiation because an internal object must share the atomic set of its creator. As usual with type-based approaches, the bindings created by aliasing cannot be modified after creation.

3.2 Arrays

Arrays are fully handled by our implementation. Supporting arrays requires being able to specify atomicity constraints at three different levels. The declaration

```
atomic(a) B[] vals;
```

indicates that the reference to array vals is part of atomic set a, however the contents of the array can be updated without synchronization. The declaration

```
atomic(a) B[] vals|this.a[]|;
```

indicates that not only is the reference to the array to be accessed atomically, but the contents of the array are also part of atomic set a and must be accessed in a synchronized manner. Finally, the declaration

```
atomic(a) B[] vals|this.a[]b=this.a|;
```

indicates that, additionally, the atomic set b of each of the objects contained within the array should be merged with atomic set a. In our experience, we found all three of these forms of array annotation to be useful.

3.3 Data Races and Deadlocks

AJ does not completely prevent programmer errors. Data races can occur within a unit of work if the code manipulates data that is not part of the unit's atomic set. Thus it is incumbent on the programmer to correctly annotate all fields which share a consistency property, and to place unitfor annotations on method parameters as needed. Forgetting to annotate a field or method parameter can result in concurrency errors.

Our implementation of atomic set associates locks with atomic sets. There is thus the potential for deadlocks when multiple non-aliased atomic sets are manipulated by the same unit of work. We support a form of deadlock avoidance for methods that have unitfor annotations, but cannot prevent deadlock when a unit of work for some atomic set a (transitively) invokes a unit of work for another atomic set b. In this respect, AJ programs are neither more nor less prone to deadlock than standard Java programs that acquire multiple locks out of order. We do, however, believe that the declarative nature of synchronization annotations in AJ simplifies the design of static analyses for detecting possible deadlocks.

4 A Formal Account of AJ

We formalize AJ in a core calculus in the style of [25], which is an idealized version of Java extended with some of the key features of our proposal. The goal of the formalization is to prove soundness of the type system and illustrate its key properties. To this end, we focus on the essential features of AJ, namely atomic sets, atomic annotations on fields, alias annotations and internal types. For simplicity, we restrict the formalization to a single atomic set per class, and exclude unitfor annotations. While both are important, they do not affect the type system which tracks aliases and internal classes. Adding multiple atomic sets would require a small change to the semantics which currently uses the addresses of objects as identifiers for atomic sets (instead, fresh values would have to be created for each atomic set). Adding unitfor would only require more complex traces. For brevity we omit orthogonal features of Java such as interfaces, exceptions, final variables, primitive data types, arrays, generics, and thread creation and thread death. We start with a presentation of the syntax (Section 4.1), static and dynamic semantics (sections 4.2 and 4.3 resp.). Section 4.4 establishes standard properties of the type system. The concurrency-control policy enforced by AJ is specified in Section 4.5 and a proof of atomic-set serializability is given in Section 4.6.

4.1 Syntax

The AJ syntax is given in Fig. 7. In our core calculus all fields are strongly private and methods are public. Without loss of generality, we use a "named form," where the

p	::= $\overline{cd}$	*program*		τ	::= C|a = this.b| | C	*type*
cd	::= ι class C extends D { as $\overline{fd}$ $\overline{md}$ }	*class*		α	::= atomic (a) | ϵ	
as	::= atomicset a | ϵ			ι	::= internal | ϵ	
fd	::= α τ f	*field*				
md	::= τ m ($\overline{\tau\ \mathsf{x}}$) { $\overline{\tau\ \mathsf{z}}$; s;return y }	*method*		E	::= [] | $E[x : \tau]$	*type env*
s	::= s;s | skip | x =this.f | x =(τ)y | this.f =z | x =new τ () | x =y.m ($\overline{\mathsf{z}}$)	*statement*				

Fig. 7. AJ's syntax. C, D are class names, f, m are field and method names, and x, y, z are names of variables or parameters. this is a distinguished variable. For simplicity, we assume that names of classes, fields, methods and variables are unique.

results of fields and variable accesses, method calls and instantiations must be immediately stored in a variable. A further simplification is the elimination of implicit upcasts for arguments, return values, and assignments. All casts are performed explicitly by cast statements which simplifies the other rules as they can assume type equality. Downcasts are safe in AJ because, as in Java, there is a runtime test to check that the object belongs to the target type and all AJ-specific properties are preserved by subtyping, i.e. subtypes have the same atomic sets and are internal if their parent is internal. Upcasts are more interesting as they involve loss of type information. For brevity, we assume the existence of a well-formed class-table CT. Auxiliary functions are given in Fig. 8. We use the shorthand $\overline{\mathsf{x}} <: \overline{\tau}$ to denote the pointwise subtype relation $x_1 <: \tau_1, \ldots, x_n <: \tau_n$. The subtyping relation is standard with the exception of the rule for types with alias annotations, which restricts subtyping to be annotation invariant.

$$\frac{\mathsf{C} <: \mathsf{D}}{\mathsf{C|a{=}this.b|} <: \mathsf{D|a{=}this.b|}}$$

We define the viewpoint adaption predicate *adapt* such that the value of $adapt(\tau, \tau')$ is the view of type τ from type τ'. If τ is a raw type C, then it is unchanged. If τ has an alias annotation, such as C|a = this.b|, and it is viewed from a type D|b = this.c|, then the value of this.b is substituted with this.c, yielding C|a = this.c|. In cases where *adapt* is undefined a type error will be reported as the type is not accessible from that particular viewpoint.

$$adapt(\mathsf{C}, \tau) = \mathsf{C}$$
$$adapt(\mathsf{C|a{=}this.b|}, \mathsf{D|b{=}this.c|}) = \mathsf{C|a{=}this.c|}$$

4.2 Type System

Classes, fields, and methods. A *class* definition C is well-typed if its fields are well-typed in the context of C. Furthermore, all methods (including non-overridden inherited methods) must be well-typed. In case the class inherits an atomic set, then it is not allowed to define a new one. If the class is declared internal it must have an atomic set, or inherit one. Finally, internal annotations must be preserved by inheritance. In the definitions below, we use the notation C *has* a to indicate that class C declares or inherits

Subtyping:

$$\frac{}{\mathsf{C} <: \mathsf{C}} \quad \frac{\mathsf{C}\ \text{extends}\ \mathsf{D}}{\mathsf{C} <: \mathsf{D}} \quad \frac{\mathsf{C} <: \mathsf{C}' \quad \mathsf{C}' <: \mathsf{D}}{\mathsf{C} <: \mathsf{D}}$$

$$\frac{\mathsf{C} <: \mathsf{D}}{\mathsf{C}|\mathsf{a}{=}\mathsf{this.b}| <: \mathsf{D}|\mathsf{a}{=}\mathsf{this.b}|}$$

Extends:

$$\frac{CT(\mathsf{C}) = \iota\ \mathsf{class}\ \mathsf{C}\ \mathsf{extends}\ \mathsf{D}\ \{\ as\ \overline{fd}\ \overline{md}\ \}}{\mathsf{C}\ \mathsf{extends}\ \mathsf{D}}$$

Type lookup:

$$\frac{\tau\ \mathsf{m}(\overline{\tau_{\mathsf{x}}\ \mathsf{x}})\ \{\ \overline{\tau_{\mathsf{z}}\ \mathsf{z}};\ \mathsf{s};\ \mathsf{return}\ \mathsf{y}\ \} \in methods(\mathsf{C})}{typeof(\mathsf{C.m}) = \overline{\tau_{\mathsf{x}}} \rightarrow \tau}$$

$$\frac{\tau\ \mathsf{f} \in fields(\mathsf{C})}{typeof(\mathsf{C.f}) = \tau}$$

Method lookup:

$$\frac{\tau\ \mathsf{m}(\overline{\tau_{\mathsf{x}}\ \mathsf{x}})\ \{\ \overline{\tau_{\mathsf{z}}\ \mathsf{z}};\ \mathsf{s};\ \mathsf{return}\ \mathsf{y}\ \} \in methods(\mathsf{C})}{mbody(\mathsf{C.m}) = (\overline{\tau_{\mathsf{x}}\ \mathsf{x}};\ \overline{\tau_{\mathsf{z}}\ \mathsf{z}};\ \mathsf{s};\ \mathsf{return}\ \mathsf{y})}$$

Local vars:

$$\frac{\begin{array}{c} H(F(\mathsf{this})) = \mathsf{C}|\omega|(\overline{r'}) \\ mbody(\mathsf{C.m}) = (\overline{\tau_{\mathsf{x}}\ \mathsf{x}};\ \overline{\tau_{\mathsf{z}}\ \mathsf{z}};\ \mathsf{s};\ \mathsf{return}\ \mathsf{y}) \\ E \equiv \overline{\mathsf{x} : \tau_{\mathsf{x}}}, \overline{\mathsf{z} : \tau_{\mathsf{z}}}, \mathsf{this} : \mathsf{C} \end{array}}{locals(\mathsf{m}, F) = E}$$

Internal lookup:

$$\frac{CT(\mathsf{C}) = \mathsf{internal}\ \mathsf{class}\ \mathsf{C}\ \mathsf{extends}\ \mathsf{D}\ \{\dots\}}{\mathsf{C}\ is\ \mathsf{internal}}$$

Fields lookup:

$$\frac{}{fields(\mathsf{Object}) = \epsilon}$$

$$\frac{\begin{array}{c} CT(\mathsf{C}) = \iota\ \mathsf{class}\ \mathsf{C}\ \mathsf{extends}\ \mathsf{D}\ \{\ as\ \overline{fd}\ \overline{md}\ \} \\ fields(\mathsf{D}) = \overline{fd'} \end{array}}{fields(\mathsf{C}) = \overline{fd'}\ \overline{fd}}$$

Methods lookup:

$$\frac{}{methods(\mathsf{Object}) = \epsilon}$$

$$\frac{\begin{array}{c} CT(\mathsf{C}) = \iota\ \mathsf{class}\ \mathsf{C}\ \mathsf{extends}\ \mathsf{D}\ \{\ as\ \overline{fd}\ \overline{md}\ \} \\ methods(\mathsf{D}) = \overline{md'} \quad \overline{md''} = \overline{md'} - \overline{md} \end{array}}{methods(\mathsf{C}) = \overline{md}\ \overline{md''}}$$

Valid Method overriding:

$$\frac{\begin{array}{c} typeof(\mathsf{C.m}) = \overline{\tau'} \rightarrow \tau'\ implies \\ \overline{\tau} = \overline{\tau'}\ and\ \tau = \tau' \end{array}}{override(\mathsf{m}, \mathsf{C}, \overline{\tau} \rightarrow \tau)}$$

Atomic set lookup:

$$\frac{\begin{array}{c} CT(\mathsf{C}) = \iota\ \mathsf{class}\ \mathsf{C}\ \mathsf{extends}\ \mathsf{D}\ \{\ as\ \overline{fd}\ \overline{md}\ \} \\ as = \epsilon \qquad \mathsf{D}\ has\ \mathsf{a} \end{array}}{\mathsf{C}\ has\ \mathsf{a}}$$

$$\frac{\begin{array}{c} CT(\mathsf{C}) = \iota\ \mathsf{class}\ \mathsf{C}\ \mathsf{extends}\ \mathsf{D}\ \{\ as\ \overline{fd}\ \overline{md}\ \} \\ as = \mathsf{atomicset}\ \mathsf{a} \end{array}}{\mathsf{C}\ has\ \mathsf{a}}$$

Atomic lookup:

$$\frac{\mathsf{atomic(a)}\ \tau\ \mathsf{f} \in fields(\mathsf{C})}{\mathsf{C.f}\ is\ \mathsf{atomic}}$$

Fig. 8. Auxiliary definitions.

an atomic set a. Atomic sets referred to in *field* declarations must exist. Checking a *method* requires typing its body in an environment E constructed by composing the disjoint sets of parameters, $\bar{\mathsf{x}}$, local variables, $\bar{\mathsf{z}}$ and the distinguished variable this. If class C has an atomic set, the type of this is $\mathsf{C}|\mathsf{a} = \mathsf{this.a}|$; This is the default case when an object is in charge of its own synchronization (i.e., its atomic set has not been aliased) and is needed to ensure that *adapt* is defined. The type of the local variable y appearing in the return statement must match the return type of the method, and if this method overrides an inherited method, the signature must be unchanged.

(T-CLASS)

$$\frac{\begin{array}{c} \overline{fd}\ \text{OK in}\ \mathsf{C} \quad methods(\mathsf{C}) = \overline{md'} \quad \overline{md'}\ \text{OK in}\ \mathsf{C} \quad (\mathsf{D}\ has\ \mathsf{a}\ \ implies\ \ as = \epsilon) \\ (\iota = \mathsf{internal}\ \ implies\ \ \mathsf{C}\ has\ \mathsf{a}) \quad (\mathsf{D}\ is\ \mathsf{internal}\ implies\ \iota = \mathsf{internal}) \end{array}}{\iota\ \mathsf{class}\ \mathsf{C}\ \mathsf{extends}\ \mathsf{D}\ \{\ as\ \overline{fd}\ \overline{md}\ \}\ \text{OK}}$$

$$\frac{(\tau \equiv \mathsf{D}|\mathsf{a}{=}\mathsf{this.b}|\ \mathit{implies}\ \mathsf{D}\ \mathit{has}\ \mathsf{a}\ \mathit{and}\ \mathsf{C}\ \mathit{has}\ \mathsf{b}) \quad (\alpha = \mathsf{atomic}\,(\mathsf{a})\ \mathit{implies}\ \ \mathsf{C}\ \mathit{has}\ \mathsf{a})}{\alpha\,\tau\,\mathsf{f}\quad \text{OK in } \mathsf{C}}\ \text{(T-FIELD)}$$

$$\frac{E \equiv \overline{\mathsf{x} : \tau_{\mathsf{x}}}, \overline{\mathsf{z} : \tau_{\mathsf{z}}}, \mathsf{this} : \tau_{\mathsf{this}} \quad E\ \vdash\ \mathsf{s}; \mathsf{return}\,\mathsf{y} \quad E(\mathsf{y}) = \tau \quad \mathsf{C}\ \mathsf{extends}\ \mathsf{D} \qquad (\mathit{if}\ \mathsf{C}\ \mathit{has}\ \mathsf{a}\ \ \mathit{then}\ \ \tau_{\mathsf{this}} \equiv \mathsf{C}|\mathsf{a}{=}\mathsf{this.a}|\ \ \mathit{else}\ \ \tau_{\mathsf{this}} \equiv \mathsf{C}) \quad \mathit{override}(\mathsf{m}, \mathsf{D}, \overline{\tau_{\mathsf{X}}} \to \tau)}{\tau\,\mathsf{m}(\overline{\tau_{\mathsf{x}}\,\mathsf{x}})\,\{\,\overline{\tau_{\mathsf{z}}\,\mathsf{z}};\ \mathsf{s}; \mathsf{return}\,\mathsf{y}\,\}\quad \text{OK in } \mathsf{C}}\ \text{(T-METHOD)}$$

Observant readers will note that we are checking inherited methods with the type of this bound to subclass C. This prevents the implicit upcast in method invocation from being used to subvert the type system. Consider the following program which, without the above treatment of inherited methods, would leak a reference to an internal object.

```
class Id extends Object {
   Id id() { Id x; x = this; return x }
}

internal class E extends Id {
   atomicset a;
}

class C extends Object {
   atomicset b;
   Id m() {
      E|a=this.b| y;   Id z;
      y = new E|a=this.b|();   z = y.id();
      return z;
} }
```

The instance of E is an internal class and should remain private to its owner (an instance of class C). Yet, if the invocation of id() were allowed, it would be possible to pass off the E object as an Id which is not protected. In our type system the assignment x=this does not type check in the context of class E. This problem is standard in ownership type systems. One could avoid type-checking inherited methods repeatedly by declaring inherited methods *anonymous*, i.e., that they do not leak the this reference [22] or inferring the property by whole program analysis as in [11]. In AJ, the only methods that need this are methods inherited by an internal class.

Statements. There are two type rules for object creation. The first rule, (T-NEW-RAW), covers the case where the object being created is not annotated with an alias. If class C has an atomic set, this means we are requesting the construction of an object that can take care of its own synchronization. The only restriction that must be enforced in this case is that the class not be declared internal as internal classes always depend on an owner. The second rule, (T-NEW-ASET), covers the case when a C object is created with an alias $|\mathsf{a} = \mathsf{this.b}|$. In which case, we check that C indeed has an atomic set a and that this refers to an object which has an atomic set b.

$$\frac{E(\mathsf{x}) = \mathsf{C} \qquad \mathsf{C}\ \mathit{not}\ \mathsf{internal}}{E\ \vdash\ \mathsf{x} = \mathsf{new}\ \mathsf{C}()}\ \text{(T-NEW-RAW)} \qquad \frac{E(\mathsf{x}) = \mathsf{C}|\mathsf{a}{=}\mathsf{this.b}| \qquad \mathsf{C}\ \mathit{has}\ \mathsf{a} \quad E(\mathsf{this})\ \mathit{has}\ \mathsf{b}}{E\ \vdash\ \mathsf{x} = \mathsf{new}\ \mathsf{C}|\mathsf{a}{=}\mathsf{this.b}|()}\ \text{(T-NEW-ASET)}$$

There are three type rules for upcasts. (T-CAST-PLAIN) covers the case where both types have no alias annotations. Rule (T-CAST-ASET) allows annotation invariant upcasts.

Finally, (T-CAST-OFF) strips the annotation from a type. This is only allowed for non-internal classes. The rule for method calls, (T-CALL), checks the types of the arguments and the return type. Viewpoint adaption is necessary to ensure that the types of the arguments and the return value are visible from the viewpoint of the receiver.

$$\text{(T-CAST-PLAIN)}\quad \frac{E(\mathsf{x}) = \mathsf{D} \quad E(\mathsf{y}) = \mathsf{C} \quad \mathsf{D} <: \mathsf{C}}{E \vdash \mathsf{y} = (\mathsf{C})\mathsf{x}}$$

$$\text{(T-CAST-ASET)}\quad \frac{E(\mathsf{x}) = \mathsf{D}|\mathsf{a}{=}\mathsf{this.b}| \quad E(\mathsf{y}) = \mathsf{C}|\mathsf{a}{=}\mathsf{this.b}| \quad \mathsf{C}\ \mathit{has}\ \mathsf{a} \quad E(\mathsf{this})\ \mathit{has}\ \mathsf{b} \quad \mathsf{D} <: \mathsf{C}}{E \vdash \mathsf{y} = (\mathsf{C}|\mathsf{a}{=}\mathsf{this.b}|)\mathsf{x}}$$

$$\text{(T-CAST-OFF)}\quad \frac{E(\mathsf{x}) = \mathsf{C}|\mathsf{a}{=}\mathsf{this.b}| \quad \mathsf{C}\ \mathit{not}\ \mathsf{internal} \quad E(\mathsf{y}) = \mathsf{C}}{E \vdash \mathsf{y} = (\mathsf{C})\mathsf{x}}$$

$$\text{(T-CALL)}\quad \frac{E(\mathsf{y}) = \tau_{\mathsf{y}} \quad \mathit{typeof}(\tau_{\mathsf{y}}.\mathsf{m}) = \overline{\tau} \rightarrow \tau \quad E(\overline{\mathsf{z}}) = \overline{\tau_{\mathsf{z}}} \quad \overline{\tau_{\mathsf{z}}} = \mathit{adapt}(\overline{\tau}, \tau_{\mathsf{y}}) \quad \tau' = \mathit{adapt}(\tau, \tau_{\mathsf{y}}) \quad E(\mathsf{x}) = \tau'}{E \vdash \mathsf{x} = \mathsf{y}.\mathsf{m}(\overline{\mathsf{z}})}$$

Consider for instance calls (1) and (2) to method m() in the example below. The return type of m is $\tau \equiv \mathsf{C}|\mathsf{c} = \mathsf{this.a}|$. At (1) $\tau_{\mathsf{y}} \equiv \mathsf{A}|\mathsf{a} = \mathsf{this.b}|$, the value of $\mathit{adapt}(\tau, \tau_{\mathsf{y}}) = \mathsf{C}|\mathsf{c} = \mathsf{this.b}|$ indicating, as expected, that the C object shares the same atomic set as the receiver. On the other hand, a2 is created with its own atomic set. Thus, at (2), the result of $\mathit{adapt}(\tau, \mathsf{A})$ is undefined. The call does not type check because it would return a value with an unknown alias.

```
class A extends Object {          class B extends Object {
  atomicset a;                      atomicset b;
  C|c=this.a| m(){                  A f() {
    C|c=this.a| x;                    A|a=this.b| a1; C|c=this.b| c1; A a2;
    x=new C|c=this.a|();              a1 = new A|a=this.b|();
    return x;                         c1 = a1.m();   //(1) OK
  }                                   a2 = new A();
}                                     c1 = a2.m();   //(2) ERROR
class C extends Object {              return a2;
  atomicset c;                      }
}                                 }
```

The rules for field selection and update check that the type of the field matches that of the variable it is stored into.

$$\text{(T-SELECT)}\quad \frac{E(\mathsf{this}) = \tau \quad E(\mathsf{x}) = \tau_{\mathsf{f}} \quad \mathit{typeof}(\tau.\mathsf{f}) = \tau_{\mathsf{f}}}{E \vdash \mathsf{x} = \mathsf{this.f}} \qquad \text{(T-UPDATE)}\quad \frac{E(\mathsf{this}) = \tau \quad E(\mathsf{y}) = \tau_{\mathsf{f}} \quad \mathit{typeof}(\tau.\mathsf{f}) = \tau_{\mathsf{f}}}{E \vdash \mathsf{this.f} = \mathsf{y}}$$

4.3 Dynamic Semantics

We formulate AJ's dynamic semantics as a small-step operational semantics. See Fig. 9 for syntax. An AJ configuration $H; \overline{T}$ consists of a single heap H of locations mapped to objects and a collection of threads $\overline{T}$. Each thread T has its own stack S, plus a unique thread id denoted ρ. A stack S is a sequence of triples $\langle \mathsf{m}\, F\ \ \mathsf{s} \rangle$ consisting of a method name m, a stack frame F mapping variables to locations, and a statement s.

$H ::= [] \mid H[r \mapsto v]$	*heap*	$F ::= [] \mid F[\mathsf{y} \mapsto r]$	*stack frame*
$T ::= \rho\, S \mid \rho\, \mathsf{NPE}$	*thread*	$v ::= \mathsf{C}\vert\omega\vert(\overline{r})$	*object*
$S ::= \epsilon \mid S\, \langle \mathsf{m}\, F\ \mathsf{s} \rangle$	*stack*	$\omega ::= r \mid \epsilon$	*owner atomic set*

Fig. 9. Syntax for heaps, threads, stacks, frames and objects.

At run-time, an object $\mathsf{C}|\omega|(\overline{r})$, consists of a class C, an atomic set owner ω (either a location r or empty) and values $\overline{r}$ for the object's fields (either locations or null).
We model multi-threaded Java programs with a fixed set of threads, $\overline{T}$, each of which initially starts with a call to a run method. Threads are terminated either when the run method returns or by a null pointer exception (NPE). The reduction relation $\xrightarrow{\ell}_\rho$ represents a step of evaluation. The label ℓ describes the action and the thread identifier ρ specifies the thread that performed it. Action labels can be one of the following: $\uparrow r.\mathsf{f}$ (field select), $\downarrow r.\mathsf{f}$ (field update), $\leftarrow r.\mathsf{m}$ (method return), $\rightarrow r.\mathsf{m}$ (method call), or ϵ (empty action). Labels will be used in Section 4.5 to define traces, they record operations that may lead to a data race (reads/writes) and operations that correspond to potential unit of work boundaries (calls/returns). Basic thread-scheduling is modeled as a non-deterministic choice in (D-SCHEDULE). Each step picks one of the threads for reduction, we assume a fixed number of threads.

$$\text{(D-SCHEDULE)}\quad \frac{H; \overline{T}\,\overline{T'}\,T \xrightarrow{\ell}_\rho H'; \overline{T}\,\overline{T'}\,T'}{H; \overline{T}\,T\,\overline{T'} \xrightarrow{\ell}_\rho H'; \overline{T}\,\overline{T'}\,T'}$$

We abuse syntax a little bit and treat return y as a statement. Returning from a call implies popping the topmost frame off the stack, and capturing the return value. Upcasts and skip statements have the expected semantics.

$$\text{(D-RETURN)}\quad \frac{F(\mathsf{y}) = r \quad F(\mathsf{this}) = r'}{H; \overline{T}\,\rho\, S\, \langle \mathsf{m}'\, F'\ \mathsf{x} = \mathsf{y}'.\mathsf{m}(\overline{\mathsf{z}}); \mathsf{s}' \rangle \langle \mathsf{m}\, F\ \mathsf{return}\, \mathsf{y} \rangle \xrightarrow{\leftarrow r'.\mathsf{m}}_\rho H; \overline{T}\,\rho\, S\, \langle \mathsf{m}'\, F'[\mathsf{x} \mapsto r]\ \mathsf{s}' \rangle}$$

$$\text{(D-CAST)}\quad \frac{}{H; \overline{T}\,\rho\, S\, \langle \mathsf{m}\, F\ \mathsf{x} = (\tau)\mathsf{y}; \mathsf{s} \rangle \xrightarrow{\epsilon}_\rho H; \overline{T}\,\rho\, S\, \langle \mathsf{m}\, F[\mathsf{x} \mapsto F(\mathsf{y})]\ \mathsf{s} \rangle}$$

Field selection extracts one of the references stored in the object, while field update modifies the content of the object at the proper location. We define $H(r.\mathsf{f}_i)$ as follows: $H(r.\mathsf{f}_i) = r_i$ if $H(r) = \mathsf{C}|\omega|(r_1 \ldots r_i \ldots, r_n)$ and $\mathit{fields}(\mathsf{C}) = \mathsf{f}_1, \ldots \mathsf{f}_i \ldots, \mathsf{f}_n$.

$$\text{(D-SELECT)}\quad \frac{F(\mathsf{this}) = r \quad H(r.\mathsf{f}_i) = r_i}{H; \overline{T}\,\rho\, S\, \langle \mathsf{m}\, F\ \mathsf{x} = \mathsf{this}.\mathsf{f}_i; \mathsf{s} \rangle \xrightarrow{\uparrow r.\mathsf{f}_i}_\rho H; \overline{T}\,\rho\, S\, \langle \mathsf{m}\, F[\mathsf{x} \mapsto r_i]\ \mathsf{s} \rangle}$$

$$\text{(D-UPDATE)}\quad \frac{F(\mathsf{this}) = r \quad F(\mathsf{x}) = r_\mathsf{x} \quad H(r) = \mathsf{C}|\omega|(\overline{r}, r_i, \overline{r'}) \quad H' \equiv H[r \mapsto \mathsf{C}|\omega|(\overline{r}, r_\mathsf{x}, \overline{r'})]}{H; \overline{T}\,\rho\, S\, \langle \mathsf{m}\, F\ \mathsf{this}.\mathsf{f}_i = \mathsf{x}; \mathsf{s} \rangle \xrightarrow{\downarrow r.\mathsf{f}_i}_\rho H'; \overline{T}\,\rho\, S\, \langle \mathsf{m}\, F\ \mathsf{s} \rangle}$$

Object creation comes in three flavors. (D-NEW-PLAIN) covers the construction of plain Java objects where the owner is empty. (D-NEW-SELF) takes care of creation of an instance of a class that has an atomic set and for which no alias annotation is specified.

In this case, the owner is the newly created object itself. Lastly, (D-NEW-ALIAS) is for the construction of objects which have an alias annotation of the form |a = this.b|. For those, we look up the owner of this and set it as the owner of the newly created object.

$$\frac{\begin{array}{c}v \equiv \mathsf{C}|\epsilon|(\mathsf{null}_1 ... \mathsf{null}_n) \quad r \text{ is fresh} \quad \mathit{not}\ \mathsf{C}\ \mathit{has}\ \mathsf{a} \\ H' \equiv H[r \mapsto v] \quad |\mathit{fields}(\mathsf{C})| = n \quad F' \equiv F[\mathsf{x} \mapsto r]\end{array}}{H; \overline{T}\,\rho\, S\, \langle \mathsf{m}\, F\ \ \mathsf{x} = \mathsf{new}\, \mathsf{C}(); \mathsf{s}\rangle \xrightarrow{\epsilon}_{\rho} H'; \overline{T}\,\rho\, S\, \langle \mathsf{m}\, F'\ \ \mathsf{s}\rangle} \quad \text{(D-NEW-PLAIN)}$$

$$\frac{\begin{array}{c}v \equiv \mathsf{C}|r|(\mathsf{null}_1 ... \mathsf{null}_n) \quad r \text{ is fresh} \quad \mathsf{C}\ \mathit{has}\ \mathsf{a} \quad H' \equiv H[r \mapsto v] \\ |\mathit{fields}(\mathsf{C})| = n \quad F' \equiv F[\mathsf{x} \mapsto r]\end{array}}{H; \overline{T}\,\rho\, S\, \langle \mathsf{m}\, F\ \ \mathsf{x} = \mathsf{new}\, \mathsf{C}(); \mathsf{s}\rangle \xrightarrow{\epsilon}_{\rho} H'; \overline{T}\,\rho\, S\, \langle \mathsf{m}\, F'\ \ \mathsf{s}\rangle} \quad \text{(D-NEW-SELF)}$$

$$\frac{\begin{array}{c}H(F(\mathsf{this})) = \mathsf{D}|r'|(\overline{r}) \quad r \text{ is fresh} \quad v \equiv \mathsf{C}|r'|(\mathsf{null}_1 ... \mathsf{null}_n) \quad H' \equiv H[r \mapsto v] \\ |\mathit{fields}(\mathsf{C})| = n \quad T \equiv \rho\, S\, \langle \mathsf{m}\, F[\mathsf{x} \mapsto r]\ \ \mathsf{s}\rangle\end{array}}{H; \overline{T}\,\rho\, S\, \langle \mathsf{m}\, F\ \ \mathsf{x} = \mathsf{new}\, \mathsf{C}|\mathsf{a} = \mathsf{this.b}|(); \mathsf{s}\rangle \xrightarrow{\epsilon}_{\rho} H'; \overline{T}\, T} \quad \text{(D-NEW-ALIAS)}$$

Method calls push a new frame on the stack with local variables initialized to null and parameters bound to corresponding arguments. For brevity, null-pointer exceptions cause threads to immediately get stuck. More accurate treatment of exceptions (e.g., catch-blocks and stack unwinding) is unnecessary for the problem at hand.

$$\frac{\begin{array}{c}F(\mathsf{y}) = r \quad F(\overline{\mathsf{z}}) = \overline{r} \quad H(r) = \mathsf{C}|\omega|(\overline{r'}) \quad \mathit{mbody}(\mathsf{C.m}) = (\overline{\tau_{\mathsf{x}}\, \mathsf{x}'};\ \overline{\tau_{\mathsf{y}}\, \mathsf{y}}; \mathsf{s}'; \mathsf{return}\ \mathsf{y}') \\ F' \equiv [\overline{\mathsf{y} \mapsto \mathsf{null}}][\overline{\mathsf{x}' \mapsto r}][\mathsf{this} \mapsto r] \quad S' \equiv S\, \langle \mathsf{m}'\, F\ \ \mathsf{x} = \mathsf{y.m}(\overline{\mathsf{z}}); \mathsf{s}\rangle \langle \mathsf{m}\, F'\ \ \mathsf{s}'; \mathsf{return}\ \mathsf{y}'\rangle\end{array}}{H; \overline{T}\,\rho\, S\, \langle \mathsf{m}'\, F\ \ \mathsf{x} = \mathsf{y.m}(\overline{\mathsf{z}}); \mathsf{s}\rangle \xrightarrow{\rightarrow r.\mathsf{m}}_{\rho} H; \overline{T}\,\rho\, S'} \quad \text{(D-CALL)}$$

$$\frac{}{H; \overline{T}\,\rho\, S\, \langle \mathsf{m}'\, F[\mathsf{y} \mapsto \mathsf{null}]\ \ \mathsf{x} = \mathsf{y.m}(\overline{\mathsf{z}}); \mathsf{s}\rangle \xrightarrow{\epsilon}_{\rho} H; \overline{T}\,\rho\, \mathsf{NPE}} \quad \text{(D-CALL-NPE)}$$

4.4 Properties

We now proceed to establish preservation and progress for our type system. As usual the proofs rely on a notion of well-formed heaps, threads and configurations as well as run-time subtyping. We start with these auxiliary definitions. In a heap H, let $\mathit{owner}_H(r) = \omega$, if $H(r) = \mathsf{C}|\omega|(\overline{r})$. Let $\mathit{internal}_H(r)$ hold if $H(r) = \mathsf{C}|\omega|(\overline{r})$ and C *is* internal. τ *is raw* means that type τ is of the form C and has no alias annotation. We write τ *not raw* to denote *not* τ *is raw*.

Run-time Subtyping Relation. The run-time subtyping relation, $r <:_{r_o,H} \tau$ indicates that a reference r is an instance of type τ at run-time, in the context of a reference r_o and a heap H. Since types may contain alias annotations that refer to this, we need a reference r_o to give meaning to this. There are three cases, if $H(r)$ is null then the relation holds for all τ. If $H(r)$ is $\mathsf{C}|\omega|(\overline{r})$ then if τ is a raw type, D, the relation holds if C $<:$ D and if C is not an internal class (to prevent leaking an internal object). The

last case is if τ is an aliased type $\mathsf{D}|\mathsf{a}=\mathsf{this.b}|$ in which case we must check that r has the same owner as r_o.

$$\frac{}{\mathsf{null} <:_{r_o,H} \tau} \qquad \frac{H(r) = \mathsf{C}|\omega|(\overline{r}) \quad \mathsf{C} <: \mathsf{D} \quad \mathsf{C}\ \mathit{not}\ \mathsf{internal}}{r <:_{r_o,H} \mathsf{D}} \qquad \frac{H(r) = \mathsf{C}|\omega|(\overline{r}) \quad \mathsf{C} <: \mathsf{D} \quad \mathit{owner}_H(r) = \mathit{owner}_H(r_o)}{r <:_{r_o,H} \mathsf{D}|\mathsf{b}=\mathsf{this.a}|}$$

Notice that the runtime subtyping relation satisfies the following property. If $r <:_{r_o,H} \tau$ and $r \neq \mathsf{null}$, then if τ *is raw* then $\mathit{not\ internal}_H(r)$, and if τ *not raw* then $\mathit{owner}_H(r) = \mathit{owner}_H(r_o)$.

Well-formed configurations. A *configuration* is well-formed, written $H; \overline{T}$ is WF, if the heap and threads are well-formed and the class table is well-typed. A heap H is well-formed if it is empty or if all fields of all objects it contains are well-typed, meaning that the reference corresponding to each field is a runtime subtype of the static type of that field. A thread T is well-formed, written T is WF in H, if it is stuck on a null pointer exception, or if all of its frames are well-formed, and it satisfies the following property: for each frame on the stack, if the this reference belongs to an internal class, then there exists another frame earlier in the stack with the same owner, but that is not internal.

A frame F is well-formed if for each variable x in the domain of F, the corresponding reference is a runtime subtype of the static type of x. The rules appear in Fig. 10.

(WF-CONFIGURATION)
$$\frac{H \text{ is WF in } H \quad \overline{T} \text{ is WF in } H \quad \vdash CT}{H; \overline{T} \text{ is WF}}$$

(WF-EMPTY-HEAP)
$$\frac{}{[] \text{ is WF in } H}$$

(WF-NPE-THREAD)
$$\frac{}{\rho\, \mathsf{NPE} \text{ is WF in } H}$$

(WF-THREAD-BOT)
$$\frac{\langle \mathsf{run}\, F\ \mathsf{s}\rangle \text{ is WF in } H \quad \mathit{not\ internal}_H(F(\mathsf{this}))}{\rho\, \langle \mathsf{run}\, F\ \mathsf{s}\rangle \text{ is WF in } H}$$

(WF-THREAD)
$$\frac{\begin{array}{c}\langle \mathsf{m}\, F\ \mathsf{s}\rangle \text{ is WF in } H \quad S \equiv S'\langle \mathsf{m}'\, F'\ \mathsf{x} = \mathsf{y.m}(\overline{\mathsf{z}'}); \mathsf{s}''\rangle \\ \rho\, S \text{ is WF in } H \\ (\exists \langle \mathsf{m}''\, F''\ s''\rangle \in S\langle \mathsf{m}\, F\ \mathsf{s}\rangle,\ \mathit{not\ internal}_H(F''(\mathsf{this})) \\ \mathit{and\ owner}_H(F''(\mathsf{this})) = \mathit{owner}_H(F(\mathsf{this})))\end{array}}{\rho\, S\langle \mathsf{m}\, F\ \mathsf{s}\rangle \text{ is WF in } H}$$

(WF-HEAP)
$$\frac{(\mathsf{C}\ \mathit{has}\ \mathsf{a}\ \mathit{implies}\ \omega \neq \epsilon) \quad H' \text{ is WF in } H \quad \mathit{fields}(\mathsf{C}) = \overline{\alpha\, \tau\, \mathsf{f}} \quad \overline{r\mathsf{z} <:_{r,H} \tau}}{H'[r \mapsto \mathsf{C}|\omega|(\overline{r\mathsf{z}})] \text{ is WF in } H}$$

(WF-FRAME)
$$\frac{\mathit{locals}(\mathsf{m}, F) = E \quad E \vdash \mathsf{s} \quad \forall\, \mathsf{x} \in \mathit{dom}(F), F(\mathsf{x}) <:_{F(\mathsf{this}),H} E(\mathsf{x})}{\langle \mathsf{m}\, F\ \mathsf{s}\rangle \text{ is WF in } H}$$

Fig. 10. Well-formedness rules.

Type Soundness. We prove type soundness of AJ by showing preservation and progress. Here, preservation means that reduction of a well-formed configuration results in a well-formed configuration, and the proof of preservation states that after a step of reduction a well-formed configuration remains well-formed.

Theorem 1. *Preservation. If $H; \overline{T}\, T\, \overline{T'}$ is WF and $H; \overline{T}\, T\, \overline{T'} \xrightarrow{\ell}_{\rho} H'; \overline{T}\, \overline{T'}\, T'$, then $H; \overline{T}\, \overline{T'}\, T'$ is WF.*

We define the notion of an *active* thread as a thread that it has not stumbled on a NPE or returned from its bottommost stack frame.

Definition 1. *A thread $T \equiv \rho\, S$ is* active*, denoted $active(T)$, if $S \not\equiv$ NPE and $S \not\equiv \langle \mathsf{run}\, F \;\; \mathsf{return}\; \mathsf{y} \rangle$.*

Progress requires that if there exists an active thread in a well-formed configuration, this thread should be allowed to make a step.

Theorem 2. *Progress. If $H; \overline{T}\, T\, \overline{T'}$ is WF and $active(T)$, then $H; \overline{T}\, T\, \overline{T'} \xrightarrow{\ell}_{\rho} H'; \overline{T}\, \overline{T'}\, T'$.*

4.5 Concurrency Control

The AJ semantics is purposefully silent about synchronization to allow for different concurrency-control strategies. The implementation presented in this paper uses mutual exclusion locks, our previous work used read-write locks, and we are experimenting with a transactional implementation.

The execution of a program can be characterized by a trace t which is a sequence of events $e_1 \ldots e_n$ performed by individual threads. For any implementation of AJ, we define the concurrency-control policy as a predicate over traces. We say that any trace accepted by an implementation is *well-formed*. The current implementation disallows multiple invocations of methods on objects having the same owner to execute concurrently by associating mutual exclusion locks to atomic set instances. We formalize this with the following definition of valid event. Let an event e be a tuple $(H, \overline{T}, \ell, \rho)$ consisting of a configuration, an action label and a thread id. We say that an event is valid if it has any action label other than a method call. An event with a method call on an object of an internal class is valid. For calls to non-internal classes, an event is valid if there are no outstanding method calls of objects with the same owner in other threads.

Definition 2. *An event $e = (H, \overline{T}, \ell, \rho)$ is* valid *if and only if, when $\ell = \!\!\rightarrow r.\mathsf{m}$, $H(r) = \mathsf{C}|r'|(\overline{r})$ and C not internal then $\nexists\, \rho' S \in \overline{T}.\rho' \neq \rho$ and $\langle \mathsf{m}\, F \;\; \mathsf{s} \rangle \in S$ and $H(F(\mathsf{this})) = \mathsf{D}|r'|(\overline{\mathsf{z}})$.*

In our implementation, a well-formed trace is a trace in which every event is valid and every configuration is WF. This property, enforced by the AJ runtime system, is not sufficient in itself to prevent data races. The type system guarantees that all objects belonging to an atomic set (in particular internal objects) are accessed only through methods that are units of work for the atomic set.

4.6 Atomic-Set Serializability

Serializability of atomic set operations follows from the above restriction to valid traces (mutual exclusion of methods of non-internal classes operating on the same atomic set) and the fact that all fields labeled atomic(a), including those of internal classes, are accessed within a method of a non-internal class operating on that atomic set. Given a well-formed trace t and an event e in t, $aset_t(e)$ gives the owner atomic set accessed by e, if any.

$$
aset_t(e) = \begin{cases} r' \text{ if } e = (H, \overline{T}, \ell, \rho) \wedge \ell \in \{\uparrow r.\mathsf{f}, \downarrow r.\mathsf{f}\} \\ \quad \wedge\, H(r) = \mathsf{C}|r'|(\overline{r}) \wedge \mathsf{C.f}\ is\ \mathsf{atomic} \\ \\ \epsilon \quad \text{otherwise.} \end{cases}
$$

We introduce *unit of work identifiers*, ranged over by meta variable u, in a trace t as follows. We consider the projection of t onto each thread ρ, which is a succession of events from the same thread. By considering method calls and returns ($\rightarrow r.\mathsf{m}$, $\leftarrow r.\mathsf{m}$), we determine where units of work start and end. We assign each unit of work a unique identifier u, and update all frames in the trace t to reflect not only the method name, but also the unit of work identifier u, as follows: $\langle \mathsf{m}\, u\, F\ \ s\rangle$. Given a well-formed trace t, and an event e, $uow_t(e)$ is the unit of work to which e belongs. $uow_t(e)$ is computed by examining the call stack of the thread that performs e, finding the *first* frame on the stack with a method on an object having the same owner as $aset_t(e)$, declared in a non-internal class, and returning the unit of work identifier corresponding to this method.

$$
uow_t(e) = \begin{cases} u \text{ if } e = (H, \overline{T}\rho S, \ell, \rho) \wedge \exists \langle \mathsf{m}\, u\, F\ \ \mathsf{s}\rangle \in S\ s.t. \\ \quad owner_H(F(\mathsf{this})) = aset_t(e) \\ \quad \wedge\ not\ internal_H(F(\mathsf{this})) \\ \quad \wedge\ \nexists \langle \mathsf{m}'\, u'\, F'\ \ \mathsf{s}'\rangle \ldots \langle \mathsf{m}\, u\, F\ \ \mathsf{s}\rangle \in S \\ \qquad s.t.\ owner_H(F'(\mathsf{this})) = aset_t(e) \\ \qquad \wedge\ not\ internal_H(F'(\mathsf{this})) \\ \\ \bot \text{ otherwise} \end{cases}
$$

Lemma 1. If $e = (H, \overline{T}, \ell, \rho)$ is an event in a well-formed trace t and $aset_t(e) \neq \epsilon$, then $uow_t(e) \neq \bot$.

Proof. Let $e = (H, \overline{T}\rho S\langle \mathsf{m}'\, F'\ \ \mathsf{s}'\rangle, \ell, \rho)$. Since $aset_t(e) = r' \neq \epsilon$, we have $\ell \in \{\uparrow r.\mathsf{f}, \downarrow r.\mathsf{f}\}$, $H(r) = \mathsf{C}|r'|(\overline{r})$, and $\mathsf{C.f}\ is\ \mathsf{atomic}$. Fields can only be accessed from this, so $r = F'(\mathsf{this})$. By (WF-THREAD), we know that there exists a frame $\langle \mathsf{m}\, F\ \ s\rangle$ in S such that $owner_H(F(\mathsf{this})) = owner_H(F'(\mathsf{this})) = aset_t(e)$, and $not\ internal_H(F(\mathsf{this}))$. Therefore, $uow_t(e) \neq \bot$.

The *events of a unit of work* u in a trace t are all the events e in t such that $uow_t(e) = u$. Given a well-formed trace t and an atomic set r, we define the set of *units of work corresponding to* r as the set that contains $uow_t(e)$ for each e in t such that $aset_t(e) = r$. By Lemma 1, we know that $uow_t(e)$ is well-defined for an event e such that $aset_t(e) \neq \epsilon$, meaning each access to a location in an atomic set is performed within a unit of work corresponding to that atomic set. Since valid traces provide mutual exclusion of units of work, we obtain atomic-set serializability.

Theorem 3. *Atomic-Set Serializability. Given a well-formed trace t and an atomic set r, the events of each of the units of work corresponding to r happen serially.*

5 Implementation

We implemented a proof-of-concept AJ-to-Java compiler as an Eclipse refactoring that rewrites the original source into a new project that holds the transformed code. The type checker assumes that data-centric synchronization annotations are given as Java comments. It parses these annotations and enforces the type rules of Section 4. Type errors are reported using markers in the Eclipse editor. The compiler uses standard Java synchronized blocks to enforce exclusion for each atomic set. Each non-private method of a non-internal class acquires the locks for all atomic sets for which it is a unit of work. The transformation has four steps:
(1) Create lock fields. The compiler generates a lock field $lock_S in any class C that declares an atomic set S. Atomic sets declared in super interfaces of C will have a lock field in C unless that same atomic set is present in C's superclass. For each lock field, an accessor method getLockForS() is created.
(2) Transform constructors. Constructors of classes with atomic sets are transformed to take additional parameters that are the lock objects to use. For classes that declare atomic sets, the constructors assign these parameters to the lock fields; for classes that inherit atomic sets, these lock objects are passed to superclass constructors.
(3) Transform object allocations to set locks. For objects not involved in alias relationships, new statements are transformed by passing a fresh lock object to the constructor. For objects in an alias relationship, the lock to use is read from the owner by calling the getLockForS() accessor method and passed to the constructor to initialize the lock field.
(4) Transform units of work to acquire all needed locks. This involves taking the lock of the atomic set of the declaring class and the locks for the atomic sets of any unitfor parameters. If only a single lock is required, a single synchronized block suffices. However, when multiple locks are needed, they must be acquired without inducing unnecessary deadlock. This is accomplished by ordering: each lock object is given an id when allocated, and locks are acquired in order of increasing id. There is a minor complication here: when the type of the argument is too general to denote an atomic set unambiguously, a unitfor must be used that omits the name of the atomic set (this situation arises, e.g., for the argument of equals() methods). To this end, each class with atomic sets implements an interface Atomic, which declares a method getLock() that returns the lock for its atomic set.

A few straightforward optimizations were implemented. If the compiler can determine that all members of an atomic set accessed in a method and in any methods it may call are final, then it will not introduce locking code. Furthermore, all transformed methods have two versions, one with locking code and one without; when the compiler can determine that any needed lock is already held, it will call the version that does not take locks. Currently, this is done for calls on `this` and calls on objects annotated to be part of an atomic set of `this`.

A limitation of our prototype is that it currently only supports one atomic set per class. Furthermore, it does not yet handle generics and nested classes. We emphasize that this is not a limitation of the approach, but an engineering tradeoff. With Eclipse's rudimentary support for AST manipulation handling those features would entail a considerable effort. Therefore, when these features are encountered in Java code to be used in AJ, we perform manual refactorings to side-step the problem. Generics are elimi-

nated by removing type parameters and replacing occurrences of these type parameters with type Object. Nested classes are dealt with in two steps. First, any non-static nested class is changed into a static nested class by introducing an explicit pointer to the surrounding object. Then, the nested classes are changed into top-level classes.

We also experimented with an alternative implementation, based on reentrant locks from java.util.concurrent but found the performance inferior to the current implementation that is based on synchronized blocks.

6 Empirical Evaluation

We report on experiments using AJ. All measurements were taken on a 2.6.32.3 Linux workstation with a 2.3GHz Intel Xeon 8-way processor and 8GB of RAM, using Sun's Java 1.6.0_17.

6.1 Java Collections Framework

As a first experiment, we investigate the effort involved in using atomic sets to create properly synchronized versions of representative classes from the Java Collections Framework. Specifically, we selected ArrayList, LinkedList, HashMap, LinkedHashMap, LinkedHashSet, HashSet, and TreeMap from package java.util in Sun's JDK 1.5 class libraries, along with any types on which these classes transitively depend. Each of these classes depends on several supertypes as well as several auxiliary classes (e.g., TreeMap declares nested classes SubMap and EntryIterator, as well as several anonymous nested classes). In total we included 63 types comprising 10,338 LOC.

Determining the placement of atomicset and atomic annotations was straightforward. The collection classes we consider are comprised of 5 distinct inheritance sub-hierarchies, and we introduce one atomic set in each of the types Collection, Map, Iterator, LinkedList_Entry, and Map_Entry, which are the roots of these sub-hierarchies. All instance fields were added to the atomic set that we introduced for the sub-hierarchy in which its declaring class occurs. This is accomplished by adding an atomic annotation to the class declaration. We placed unitfor annotations on constructors that take other collections as an argument, on "bulk" methods such as addAll(), and on equals() methods in order to avoid concurrency bugs that could otherwise arise if the collection object that is passed as an argument is modified concurrently during the manipulation of the collection object pointed to by this. Such concurrency-related bugs are problematic in the Java Collections Framework [8,23,12] and our approach completely avoids them.

Introducing alias annotations required somewhat more thought, as this involves atomic sets in two classes. For example, the allocation of an AbstractList_ListItr object in class AbstractList was annotated as follows: new AbstractList_ListItr |l=this.L|(...), indicating that atomic set l in the newly created iterator-object is aliased with atomic set L in the list pointed to by this. Of the classes we annotated, only LinkedList_Entry was made internal. Map_Entry could not be made internal because it is exposed to client code via methods such as Map.entrySet() that provide a direct view on the map. Our type system prohibits this as internal types cannot be returned by public methods.

type	#
atomicset	0
atomic class	5
atomic	0
unitfor	55
alias	330
array object	24
array element	16
TOTAL	430

(a)

type	#
atomicset	1
atomic class	14
atomic	25
unitfor	0
alias	8
array object	0
array element	1
TOTAL	49

(b)

Table 1. **(a)** Number of annotations required for annotating 63 classes (10,860 LOC) from the Java Collections Framework. **(b)** Number of annotations required for SPECjbb (7,891 LOC). 125 synchronized annotations could be removed in SPECjbb.

The introduction of annotations required a few minor textual code changes. In particular, atomic fields must be accessed through accessor methods. Making LinkedList_Entry internal caused the LinkedList.addBefore() method to be rejected by our type-checker as it returned an internal class. This method could not be made private because it was invoked by LinkedList_ListItr.add(). However, as add() ignored the return value of this method call, we resolved the problem by creating a method addBefore2() with identical functionality as addBefore(), but with return type void.

Table 1(a) classifies the annotations in the 63 annotated classes. As the table shows, we need a total of 430 annotations in 63 classes comprising 10,860 LOC. The majority of these annotations are related to ownership (aliasing), due to the pervasive use of iterators and auxiliary data structures such as list entries. This amounts to approximately 40 annotations per KLOC of source code, which is somewhat higher than the annotation overhead of the type systems by Flanagan et al. that guarantee race-freedom [8,1] or atomicity [7]. However, in our case, we *generate* properly synchronized code and guarantee serializability from these annotations alone, whereas Flanagan et al. require a program that is *already synchronized* using Java's synchronized construct.

6.2 SPECjbb

We refactored a widely used multi-threaded performance benchmark, SPECjbb,[2] to use atomic sets. SPECjbb simulates a server-side application with classes representing entities like companies, customers, warehouses, and performing activities such as generating orders and making deliveries. Customers are represented by driver threads and database storage is simulated using the TreeMap binary tree class. SPECjbb uses synchronized statements and methods for ensuring mutual exclusion during order processing and wait()/notify() for coordinated ramp up and shut down of threads. We studied the existing synchronization in SPECjbb's source code in order to understand how atomic sets could be introduced. In the course of this analysis, we observed several issues:

Inconsistent synchronization. Synchronization appears to be somewhat haphazard. For instance, class Customer initializes shared fields in its constructor and in set-

[2] http://www.spec.org/jbb2005

UsingRandom(). Some of these fields have synchronized accessors, whereas others, like address, have unsynchronized accessors. Several methods (e.g., TreeMapDataStorage.deleteFirstEntities()) should logically be executed atomically, but there is no synchronization to enforce this.

Redundant synchronization. Many accessor methods in class Stock are synchronized even though the accessed fields are written only once, in a method called only by the constructor (e.g., Stock.getId()).

Use of wait/notify. The wait() and notify() methods are used to implement barriers that coordinate the threads of the multiple warehouses so that they ramp up, run and shut down in a synchronized manner. According to monitor semantics, when a thread calls wait(), it releases its receiver object's lock and is suspended until it is "woken up" when another thread calls notify() on the same object.

Ownership issues. Several data structures rely on collections from the Java Collections Framework to store data. For example, TreeMapDataStorage relies on a TreeMap to store its data. As mentioned, several methods of this class (e.g., deleteFirstEntities()) should logically be executed atomically but do contain synchronization to achieve this.

Our approach was to add atomic sets in a straightforward way. Since we did not know the exact semantics of SPECjbb and the benchmark does not perform meaningful self-checking, we assumed that it was correct and verified that any synchronized section in the original code would be a unit of work in the AJ version. This check was done manually, by comparing the transformed AJ code to the original benchmark. The atomic set annotations solved the issue of inconsistent synchronization mentioned above, as all accesses to fields that are part of an atomic set are guaranteed to be protected. For the ownership issue related to collections, our code reused the AJ versions of the collections of Section 6.1. Dealing with wait()/notify() required a bit of work as this idiom is not supported by our model and care is required to avoid deadlocks when calling wait(). We chose to break up units of work that contain a wait() call in two distinct units of work, one for the code before the call and one for the code after, and leave the call to wait() in a non-unit-of-work body.

Table 1(b) shows the number of annotations required for creating the resulting *basic* AJ version of SPECjbb. Only 49 annotations were required, that is approximately 6.2 annotations per KLOC. These annotations replace 125 occurrences of synchronized in the original code.[3] The annotation overhead is significantly lower than for the Java Collections Framework because very few ownership issues occurred in SPECjbb that require the use of aliasing.

After obtaining the *basic* AJ version of the benchmark, we investigated how to improve its performance by decreasing the number of synchronization operations performed at run time. After some profiling, we identified two small improvements that could be easily applied to the code to yield a *tuned* version. First, we found that SPECjbb does not use any Map.Entry objects, which allowed us to refactor the java.util library from the previous experiment to make this interface and its subclasses internal. Second, SPECjbb contains several transactions that iterate over a thread-local collection.

[3] A small number of occurrences of synchronized remain, they are related to uses of wait() and to synchronization in static methods.

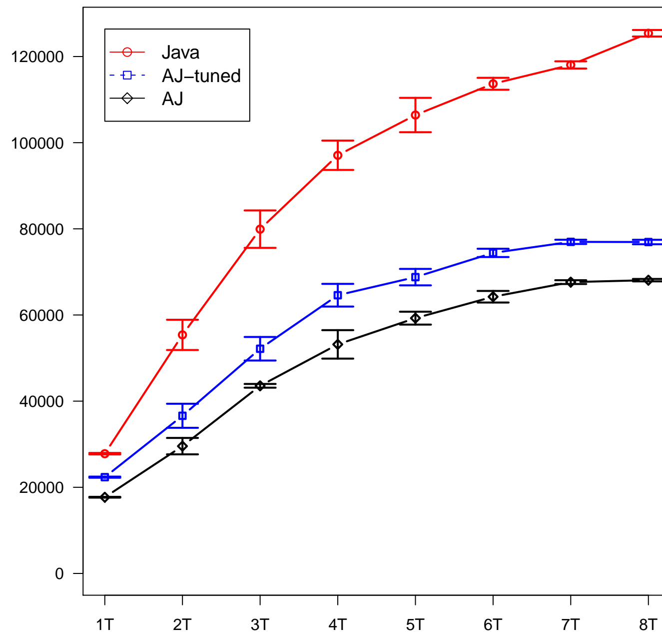

Fig. 11. Performance measurements for SPECjbb. The figure shows the average number of bops (a measure of throughput, higher is better) and standard deviation achieved by the *original* code, and by the *basic* and *tuned* AJ implementations, for up to 8 threads.

However, the basic AJ version acquired a lock for each hasNext() and next() call. We refactored these loops into a new method where the Iterator is a unitfor parameter, so that the Iterator's lock is acquired only once and all the above calls use the internal version which omits locking. Our benchmarking demonstrates that these two changes result in a substantial speed up. The *tuned* version of SPECjbb requires 3 additional unitfor annotations; the numbers of the other types of annotations remain as in Table 1(b).

Figure 11 compares the performance of the original implementation of SPECjbb to that of the *basic* and *tuned* AJ implementations. It reports the average and standard deviation in SPECjbb2005 bops, which is a measure of the number of transactions per second, obtained from 10 series of 2-minute runs with increasing numbers of threads (ranging from 1 to 8) for each version. From these measurements, it can be seen that, for a single thread, the *basic* AJ implementation of SPECjbb achieves a throughput of approximately 63.6% of that of the *original* implementation. The *tuned* implementation performs better, reaching 80.4% of the throughput of the original implementation. The graph shows that while the AJ version scales, it does not do as well as the original SPECjbb code. Specifically, for the situation with 8 threads, we measure a throughput of 54.3% and 61.4% of that of the *original* Java version, for the *basic* and *tuned* versions, respectively. This can be attributed to some of the additional locking introduced by atomic sets and reducing this overhead is clearly an important topic for future work.

To provide a rough assessment of the cost of locking, we ran a version of SPECjbb with no synchronization in single-threaded mode. The unsynchronized version achieves 32712 bops, compared to 27801 bops for the original code and 22350 for the tuned AJ code.

7 Conclusions

We have presented a type-based approach for data-centric synchronization, based on atomic sets and units of work. Our new type system guarantees atomic-set serializability while enabling separate compilation and atomic sets that span multiple objects. We implemented this approach in AJ, a significant subset of Java extended with atomic sets, and created an AJ-to-Java compiler. We demonstrated that our approach has low annotation overhead, by manually rewriting into AJ several classes from the Java Collections Framework, and SPECjbb, a widely used multi-threaded performance benchmark. In our experiments, the annotation overhead ranged from approximately 40 annotations for each KLOC of source code in Java Collections, to only 6.2 annotations per KLOC in SPECjbb. We expect SPECjbb to be representative of the majority of user written code where concurrency concerns are only a small part of the code. As performance optimizations were not the main focus of this work we consider the reported results to be encouraging as our approach is capable of generating code with acceptable performance while providing a correctness guarantee that Java's current synchronization mechanism does not offer. In future work, we plan to explore several avenues for improving performance, including the use of program analyses to tighten the scope of synchronization. We also plan to explore the use of static analysis for detecting possible deadlock.

Acknowledgments. We are grateful to Dan Marino and the anonymous reviewers for their constructive feedback on drafts of this paper. This work is supported in part by NSF grants CCF 0938232 and CNS 0716659 as well as ONR award N000140910754.

References

1. M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems*, 28(2), 2006.
2. C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, November 2002.
3. C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2001.
4. L. Ceze, C. von Praun, C. Cascaval, P. Montesinos, and J. Torrellas. Concurrency control with data coloring. In *Workshop on Memory Systems Performance and Correctness (MSPC)*, pages 6–10, 2008.
5. S. Cherem, T. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. In *Conference on Programming Language Design and Implementation (PLDI)*, 2008.
6. D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *Conference on Object-Oriented Programming, Languages, and Applications (OOPSLA)*, 1998.

7. C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Conference on Programming Language Design and Implementation (PLDI)*, 2003.
8. C. Flanagan and S. Freund. Type-based race detection for Java. In *Conference on Programming Language Design and Implementation (PLDI)*, June 2000.
9. C. Flanagan, S. N. Freund, M. Lifshin, and S. Qadeer. Types for atomicity: Static checking and inference for Java. *ACM Transactions on Programming Languages and Systems*, 30(4), 2008.
10. A. Greenhouse and J. Boyland. An object-oriented effect system. In *European Conference on Object-Oriented Programming (ECOOP)*, 1999.
11. C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating objects with confined types. *Transactions on Programming Languages and Systems*, 29(6):32–73, 2007.
12. C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic detection of atomic-set-serializability violations. In *International Conference on Software Engineering (ICSE)*, 2008.
13. T. Harris and K. Fraser. Language support for lightweight transactions. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 388–402, November 2003.
14. M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *International Symposium on Computer Architecture (ISCA)*, 1993.
15. C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.
16. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396 – 450, May 2001.
17. N. Kidd, T. W. Reps, J. Dolby, and M. Vaziri. Finding concurrency-related bugs using random isolation. In *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2009.
18. K. Rustan M. Leino. Data Groups: Specifying the modification of extended state. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 1998.
19. B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: Synchronization inference for atomic sections. In *Symposium on Principles of Programming Languages (POPL)*, 2006.
20. J. Noble, J. Potter, and J. Vitek. Flexible alias protection. In *European Conference on Object-Oriented Programming (ECOOP)*, 1998.
21. M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *Symposium on Principles of Programming Languages (POPL)*, 2006.
22. J. Vitek and B. Bokowski. Confined types in Java. *Software Practice & Experience*, 31(6):507–532, 2001.
23. L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2006.
24. L. Wang and S. D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Transactions on Software Engineering*, 32(2), 2006.
25. T. Wrigstad, F. Pizlo, F. Meawad, L. Zhao, and J. Vitek. Loci: Simple thread-locality for Java. In *European Conference on Object Oriented Programming (ECOOP)*, July 2009.